To appear, *Proc. 14th National Conf. on AI (AAAI-97)*, Providence, August, 1997

Planning with Concurrent Interacting Actions

Craig Boutilier and Ronen I. Brafman
Computer Science Department
University of British Columbia
Vancouver, B.C., Canada V6T 1Z4
{cebly,brafman}@cs.ubc.ca

Abstract

In order to generate plans for agents with multiple actuators or agent teams, we must be able to represent and plan using concurrent actions with interacting effects. Historically, this has been considered a challenging task that could require a temporal planner. We show that, with simple modifications, the STRIPS action representation language can be used to represent concurrent interacting actions. Moreover, current algorithms for partial-order planning require only small modifications in order to handle this language and produce coordinated multiagent plans. These results open the way to partial order planners for cooperative multiagent systems.

1 Introduction

In order to construct plans for agent teams or agents with multiple actuators, such as multi-armed robots, we must be able to model the effects and interactions of multiple actions executed concurrently, and generate plans that take these interactions into account. A viable solution to the multiagent planning (MAP) problem must include economical action descriptions that are convenient to specify and are easily manipulable by planning algorithms, as well as planning methods that can deal with the interactions generally associated with concurrent actions.

Surprisingly, despite the recent interest in multiagent applications—for instance, in robotics [7, 10] and distributed AI [8]—very little research addresses the MAP problem.[2] Some authors (see, e.g., [15]) have considered the representation of concurrent actions and a number of contemporary planners can handle concurrent *noninteracting* actions to a certain degree. However, the prevailing wisdom seems to suggest that *temporal planners* are required to adequately deal with general MAP problems (see, e.g., Knoblock's discussion of this in [11]). Certainly time plays a role in planning—in any planner, the idea that sequences of actions occur embodies an implicit notion of time. However, we disagree that time in multiagent planning must be dealt with in a more explicit fashion than in single-agent planning. The main aim of this paper is to demonstrate that a form of the MAP problem can be solved using very simple extensions to existing STRIPS representations and (single-agent) planners like UCPOP [14]. We provide a representation for interacting actions and a MAP algorithm that requires no explicit representation of time.

The central issue in multiagent planning lies in the fact that individual agent actions *do* interact. Consider the following example: two agents must move a large set of blocks from one room to another. While they could pick up each block separately, a better solution would be to use an existing table in the following manner. First, the agents put all blocks on the table, then they each lift one side of the table. However, they must lift the table simultaneously; otherwise, if only one side of the table is lifted, all the blocks will fall off. Having lifted the table, they must move it to the other room. There, they put the table down. In fact, depending on the precise goal, it may be better for one agent to drop its side of the table first, causing the blocks to fall off. They might then return the table. Notice how this plan requires the agents to coordinate in two different ways: First, they must lift the table together so that the blocks do not fall; later, one of them (and only one) must drop its side of the table to let the blocks fall. An action representation that makes such interactions explicit and a planning algorithm that can, as result of these interactions, prescribe that certain actions must or must not be executed concurrently are the main features of any multiagent planner—temporal representations are not the central problem. Certainly, in multiagent domains the need to explicitly model continuous processes or time constraints may be more urgent. These issues, however, also arise in single-agent planning.

Since the actions of distinct agents interact, we cannot, in general, specify the effects of an individual's actions without taking into account what other actions might be performed by other agents at the same time. One possible solution to this problem is to specify the effects of all *joint actions* directly. More specifically, let A_i be the set of actions available to agent i (assuming n agents labeled $1 \ldots n$), and let the *joint action space* be $A_1 \times A_2 \times \cdots \times A_n$. We treat each element of this space as a separate action, and specify its effects using our favorite action representation.[3] This approach has a num-

[2] Lansky's work is one exception, but it does not build on conventional planning techniques [12].

[3] Our discussion will center on the STRIPS action representation, but similar considerations apply to other representations such

ber of drawbacks. First, the number of joint actions increases exponentially with the number of agents. Second, this fails to exploit the fact that a number of individual actions may not interact at all, or at least not interact under certain conditions. We would like a representation of actions in multiagent settings that exploits the independence of individual action effects to whatever extent possible. For instance, while the lift actions of the two agents may interact, many other actions will not (e.g., one agent lifting the table and another picking up a block). Hence, we do not need to explicitly consider all combinations of these actions, and can specify certain individual effects separately, combining the effects "as needed." Finally, the use of joint actions in the context of most planners forces what seems to be an excessive amount of commitment. Whenever, the individual action of some agent can accomplish a desired effect, we must insert into our plan a joint action, thereby committing all other agents to *specific* actions to be executed concurrently, even though the actual choices may be irrelevant. For these reasons, we desire a more "distributed" representation of actions, as in the multi-entity model of [13].

We are therefore faced with the following two problems: (1) The representation problem: how do we naturally and concisely represent the effects of actions that may be influenced (positively or negatively) by the concurrent performance of other actions (or exogenous events for that matter); (2) The planning problem: how do we plan for multiple agents using such a representation.

In this paper, we show how the STRIPS action representation can be augmented to handle concurrent interacting actions and how existing nonlinear planners can be adapted to handle such actions. In fact, it might come as a surprise that solving both problems requires only a small number of changes to existing nonlinear planners, such as UCPOP [14].[4] The main addition to the STRIPS representation for action a is a *concurrent action list*: this describes restrictions on the actions that can (or cannot) be executed concurrently in order for a to have the specified effect (indeed, a can have a number of different *conditional effects* depending on which concurrent actions are applied). In order to handle this richer language, we must make a number of modifications to "standard" partial-order planners: (a) we add equality (inequality) constraints on action orderings to enforce concurrency (non-concurrency) constraints; and (b) we expand the definition of *threats* to cover concurrent actions that could prevent an intended action effect.

In the following section we describe our STRIPS-style representation for concurrent, interacting actions. This is followed by a semantics for concurrent plans in Section 3. In Section 4 we describe the *Partial Order Multiagent Planning* algorithm (POMP), a modified version of the UCPOP algorithm that can be used to generate multiagent plans. Section 5 concludes the paper. A longer version of this paper [3] contains an examination of the use of dynamic Bayes nets for representing (possibly probabilistic) actions in multiagent domains, as well as an extensive example.

```
(define (operator pickup)
:params (?a1 ?x)
:pre    (and (inroom ?a1 ?r1) (inroom block ?r1)
             (handempty ?a1) (onfloor ?x))
:conc   (not (= ?a1 ?a2)):(not (pickup ?a2 ?x))
:eff    (and (not (handempty ?a1)
          (not (onfloor ?x))
          (holding ?a1 Block))))
```

Figure 1: The PickUp action

2 Representing Concurrent Action

We assume a basic familiarity with the STRIPS action representation: states are represented using sets (conjunctions) of positive literals, and actions are represented using *effect lists*, summarizing the effect of the action on a state. In this paper, we use a standard variant of STRIPS in which the domain theory is defined using a set Λ of action schemata with typed variables.[5] This allows for a more concise description of the set of actions, and it can be exploited by least commitment planners. By convention, each action schema will have as its first argument a free variable denoting the acting agent (this is typically not needed in single-agent domains). The STRIPS representation can be enhanced using a more expressive language. For instance, UCPOP [14] allows a form of universal quantification in the action description and conditional effects. We do not discuss quantification here, but we will consider conditional effects.

We consider a simple extension of STRIPS for actions whose effects can be influenced by the concurrent execution of other actions. To each action description we add a (possibly empty) *concurrent list*: this contains a list of action schemata, each of which may be prefixed by negations and certain codesignation and non-codesignation constraints. Intuitively, this list specifies which actions can co-occur or cannot co-occur with the given action in order to produce the effect so described. This list is treated much like a set of preconditions, although it refers to concurrently executed actions rather than conditions that must hold prior to execution.

An example action schema for the *Pickup* action is described in Figure 1. This action can be executed if: the agent and the block are in the same room; the agent's hand is empty; the block is on the floor; and no *other* agent is attempting to pickup the block concurrently. If these preconditions and concurrency conditions hold, then the block will successfully be picked up by the agent.

Using this representation, we can represent actions whose effects are modified by the concurrent execution of other actions. For example, suppose an agent a_1 can pick up one side of a table, with the effect of dumping blocks onto the floor if no agent a_2 picks up the other side, and with the effect of simply raising the table if a_2 picks up the other side. Clearly, the concurrency conditions `(not (pickup ?a2 ?s))` and `(pickup ?a2 ?s)` can be used to distinguish

as the situation calculus [16] and dynamic Bayes nets [6, 4].

[4] Moreover, other planning algorithms, (e.g., [1, 9]) should prove amenable to extension to multiagent planning using similar ideas.

[5] The general concepts and notation to follow, apart from specific multiagent extensions, draws heavily on Weld's excellent survey of partial order planning [17].

```
(define (operator lower)
:params  (?a1 ?s1)
:prec    (and (holding ?a1 ?s1) (up ?s1))
:conc    (and (not (lift ?a2 ?s2)) (not (= ?a1 ?a2))
              (not (= ?s1 ?s2)))
:eff     (and (not (up ?s1))(down ?s1)(not (holding ?a1 ?s1))
              (forall ((object ?x))
               (when (ontable ?x)(up ?s2)(not (= ?s1 ?s2))
                 (and (not (lower ?a3 ?s2))(not (= ?a3 ?a1)))
                 (and (onfloor ?x)(not (ontable x))))))
```

Figure 2: The Lower action

the two cases; but treating them as preconditions essentially splits the action into two separate actions with similar effects. As in single-agent representations, we can treat such "modifiers" using a *when* clause, essentially specifying the *conditional effects* of an action. The distinction in our case is simply that, in addition to state conditions, the antecedent of a *when* clause can refer to the concurrent execution of actions (or their negation). The syntax of concurrency constraints in *when* clauses will be like that of the concurrency list; but instead of treating them as preconditions, they will designate conditions under which the action has the effect given by the consequent. The general form of the *when* clause is `(when` *antecedent effect)*, where *antecedent* itself consists of two parts: *(conditions conc-constraints)*. The table lowering action schema is described in Figure 2. Its preconditions are that the agent is holding side $?s1$ of the table which is raised. It has a non-concurrency condition stating that no other agent is simultaneously raising the other side of the table. (Notice non-concurrency conditions are implicitly universally quantified). Its primary effect is to cause $?s1$ to be "down." In addition, the conditional effect states that when there is no concurrent lower action of the other side of the table, and there is some object on the table, that object falls to the floor.

An action description can have no *when* clause, one *when* clause, or multiple *when* clauses. In the latter case, the preconditions of all the *when* clauses must be disjoint.[6]

The semantics of individual actions is, of course, different in our multiagent setting than in the single-agent case. It is not individual actions that transform an initial state of the world into a new state of the world. Rather, it is joint actions (i.e., n-tuples of individual actions, possibly including no-ops, one for each agent) that define state transitions. Given a state s and a joint action $a = \langle a_1, \cdots, a_n \rangle$, the state t that results from performing a at s is such that all atoms in the add lists of each a_i are true in t, all atoms in the delete lists of each a_i are false in t, and all unmentioned atoms are unchanged from s. Under this semantics, an action description can be inconsistent if some individual action a causes Q to be true, and another action b causes Q to be false. If this is the case, it is the responsibility of the axiomatizer to recognize the conflict and state the true effect if a and b are performed concurrently (by imposing conditional effects with concurrent action conditions) or to disallow concurrent execution (by imposing non-concurrency conditions). We assume all action descriptions are consistent in the sequel.

Several interesting issues arise in the specification of actions for multiple agents. First, we assume throughout that each agent can perform only one action at a time, so any possible concurrent actions must be performed by distinct agents. This allows our actions descriptions to be simpler than they otherwise might. If we allowed a single agent to perform certain actions concurrently, but not others, we would have to add extra concurrency constraints that preclude actions that might be executable by another agent from being performed by the acting agent.

Another issue that must be addressed is the precise effect of a joint action, one of whose individual actions negates some precondition of a concurrently executed individual action. We make no special allowances for this, simply retaining the semantics described above. While this does not complicate the definition of joint actions, some such combinations may not make sense. Again, we could treat these in several ways: we can allow the specification of such actions and design the planner so that it excludes such combinations when forming concurrent plans, unless an explicit concurrency condition is given (this means the axiomatizer need not think about such interactions); or we can allow such combinations, in general, but explicitly exclude problematic cases by adding non-concurrency constraints.

Finally, an undesirable (though theoretically unproblematic) situation can arise if we provide "inconsistent" concurrency lists. For example, we may require action a to be concurrent with b in order to have a particular effect, while b may be required to be non-concurrent with a (this can span a set of actions with more than two elements, naturally). This simply means that we cannot really achieve the intended effect of a, and the planner will recognize this; but such a specification can lead to unnecessary backtracking during the planning process. We will generally assume that concurrency lists are consistent.

3 Representing Concurrent Plans

Before moving on to discuss the planning process, we describe our representation for multiagent plans, which is rather straightforward extension of standard single-agent partially ordered plan representations. A (single-agent) nonlinear plan consists of a set of action instances, together with various strict *ordering constraints* (i.e., using the relations $<$ and $>$) on the ordering of these actions, as well as *codesignation* and *non-codesignation constraints* on the values of variables appearing in these actions, forcing them to have the same or different values, respectively [17, 14]. A nonlinear plan of this sort represents its set of possible *linearizations*, the set of totally ordered plans formed from its action instances that do not violate any of the ordering, codesignation and non-codesignation constraints.[7] We say a nonlinear plan is *consistent* if it has some linearization. The set of linearizations can be seen as the "semantics" of a nonlinear plan in some

[6] In the case of multiple clauses, the disjointness restriction can be relaxed if the effects are independent, much like in a Bayes net action description [4]. We discuss the use of dynamic Bayes nets and the advantages they offer as an action representation method for multiagent systems in [3].

[7] Concurrent execution has also been considered in this context for non-interacting actions; see [11] for a discussion.

sense; a (consistent) nonlinear plan *satisfies* a goal set G, given starting state s, if any linearization is guaranteed to satisfy G.

A *concurrent nonlinear plan* for n agents (labeled $1, \ldots n$) is similar: it consists of a set of action instances (with agent arguments, though not necessarily instantiated) together with a set of arbitrary ordering constraints over the actions (i.e., $<, >, =$ and $\neq$) and the usual codesignation and non-codesignation constraints. Unlike single-agent nonlinear plans, we allow equality and inequality ordering constraints so that concurrent or non-concurrent execution of a pair of actions can be imposed. Our semantics must allow for the concurrent execution of actions by our n agents. To this end we extend the notion of a linearization:

Definition Let P be a concurrent nonlinear plan for agents $1, \ldots n$. An *n-linearization* of P is a sequence of joint actions $A_1, \cdots A_k$ for agents $1, \ldots n$ such that

1. each individual action instance in P is a member of some joint action A_i;
2. no individual action occurs in $A_1, \cdots A_k$ other than those in P or individual *No-op* actions;
3. the codesignation and non-codesignation constraints in P are respected; and
4. the ordering constraints in P are respected. More precisely, for any individual action instances a and b in P, and joint actions A_j and A_k in which a and b occur, any ordering constraints between a and b are true of A_j and A_k; that is, if $a\{<, >, =, \neq\}b$, then $j\{<, >, =, \neq\}k$.

In other words, the actions in P are arranged in a set of joint actions such that the ordering of individual actions satisfies the constraints, with "synchronization" ensured by no-ops. If we have a set of k actions (which are allowed to be executed by distinct agents) with no ordering constraints, the set of linearizations includes the "short" plan with a single joint action where all k actions are executed concurrently by different agents (assuming $k \leq n$), a "strung out" plan where the k actions are executed one at a time by a single agent, with all others doing nothing (or where different agents take turns), "longer" plans stretched out even further by joint no-ops, or anything in between.

The definition of n-linearization requires that no agent perform more than one action at a time. This conforms with the assumption we made in the last section, though the definition could quite easily be relaxed in this regard. Because of no-ops, our n-linearizations do not correspond to shortest plans, either in the concurrently or non-concurrently executed senses of the term. However, it is a relatively easy matter to "sweep through" a concurrent nonlinear plan and construct some *shortest n-linearization*, one with the fewest joint actions, or taking the least amount of "time." Though we do not have an explicit notion of time, the sequence of joint actions in an n-linearization implicitly determines a time line along which each agent must execute its individual actions. The fact that concurrency and non-concurrency constraints are enforced in the linearizations ensures that the plan is coordinated and synchronized. We note that in order to *execute* such a plan in a coordinated fashion the agents will need some synchronization mechanism. This issue is not dealt with in this paper.

4 Planning with Concurrent Actions

The POMP algorithm, a version of Weld's POP algorithm [17] modified to handle concurrent actions, is shown in Figure 3.[8] To keep the discussion brief, we first describe POMP without considering conditional effects.

We assume the existence of a function *MGU*(Q, R, B) which returns the most general unifier of the literals Q and R with respect to the codesignation constraints in B. The algorithm has a number of input variables: the set A contains all action instances inserted into the plan so far; the set O contains ordering constraints on elements of A; the set L contains causal links; the set *NC* contains non-concurrency constraints; and the set B contains the current codesignation constraints. The set *NC* does not appear in Weld's POP algorithm and contains elements of the form $\mathsf{A} \neq a$, where A is an action schema and a is an action instance from A. Intuitively, a non-concurrency constraint of this form requires that no action instance a' that matches the schema A, subject to the (non) codesignation constraints, appear concurrently with a in the plan.

The *agenda* is a set of pairs of the form $\langle Q, A\rangle$, each listing a precondition Q that has not yet been achieved and the action A that requires it. Initially, the sets L, *NC*, and B are empty, while A contains the two fictitious actions A_0 and A_∞, where A_0 has the initial state propositions as its effects and A_∞ has the goal state conjuncts as its preconditions. The agenda initially contains all pairs $\langle Q, A_\infty\rangle$ such that Q is a goal state conjunct. This specification of the initial agenda is identical to that used in POP [17]. Finally, we note that the *choose* operator, which appears in the **Action Selection** and **Concurrent Action Selection** steps, denotes nondeterministic choice. Again, this device is just that used in POP to make algorithm specification independent of the search strategy actually used for planning.

Many of the structures and algorithmic steps of POMP correspond exactly to those used in POP. Rather than describe these in detail, we focus our discussion on the elements of POMP that differ from POP. Apart from the additional data structure *NC* mentioned above, one key difference is the additional **Concurrent Action Selection** step in POMP, which takes care of the concurrency requirements of each newly instantiated action.

One final key distinction is the notion of a *threat* used in POMP, which is more general than that used by POP. Much like POP, given a plan $\langle A, O, L, NC\rangle$, we say that A_t *threatens* the causal link $A_p \xrightarrow{Q} A_c$ when $O \cup \{A_p \leq A_t < A_c\}$ is consistent, and A_t has $\neg Q$ as an effect. Threats are handled using *demotion* (much like in POP), or *weak promotion*. The latter differs from the standard promotion technique used in POP: it allows A_t to be ordered *concurrently* with A_c, not just after A_c.[9]

[8] A treatment of the more general UCPOP algorithm appears in [3], but is essentially similar.

[9] If we wish to exclude actions that negate some precondition of

POMP($\langle A, O, L, NC, B\rangle$,*agenda*)

Termination: If agenda is empty, return $\langle A, O, L, NC, B\rangle$.

Goal Selection: Let $\langle Q, A_{need}\rangle$ be a pair on the *agenda*. (A_{need} is an action and Q is a conjunct from its precondition list.)

Action Selection: Let A_{add} = *Choose* an action (newly instantiated or from A), one of whose effects unifies with Q subject to the constraints in B. If no such action exists, then return failure.

Let $L' = L \cup \{A_{add} \stackrel{Q}{\rightarrow} A_{need}\}$. Form B' by adding to B any codesignation constraints that are needed in order to force A_{add} to have the desired effect. Let $O' = O \cup \{A_{add} < A_{need}\}$. If A_{add} is newly instantiated, then $A' = A \cup \{A_{add}\}$ and $O' = O' \cup \{A_0 < A_{add} < A_\infty\}$ (otherwise, let $A' = A$).

Concurrent Action Selection: If A_{add} is newly instantiated then apply the following steps to every positive action α_{conc} in A_{add}'s concurrent list: Let A_{conc} = *Choose* a newly instantiated action from Λ or an action that is already in A and can be ordered consistently concurrently with A_{add}. Make sure that there is a free agent that can perform this action concurrently with A_{add} and any other concurrently scheduled actions. If no such action exists then return failure. Let $O' = O \cup \{A_{conc} = A_{need}\}$. If A_{conc} is newly instantiated, then $A' = A \cup \{A_{add}\}$ and $O' = O' \cup \{A_0 < A_{conc} < A_\infty\}$ (otherwise, let $A' = A$). If a_{add} is the agent variable in A_{add} and a_{conc} is the agent variable in A_{conc}, then add $a_{add} \neq a_{conc}$ to B', as well as all similar non-codesignation constraints for actions A such that $A = A_{add} \in O$.
Re-apply this step to A_{conc}, if needed.

For every negative action schema $A_{\neg conc}$ in A_{add}'s concurrent list let $NC' = NC \cup \{A_{\neg conc} \neq A_{add}\}$. Add to B' any codesignation constraints associated with $A_{\neg conc}$.

Updating of Goal State: Let *agenda'* = *agenda* $- \{\langle Q, A_{need}\rangle\}$. If A_{add} is newly instantiated, then add $\{\langle Q_j, A_{add}\rangle\}$ to *agenda'* for every Q_j that is a logical precondition of A_{add}. Add the other preconditions to B'. If additional concurrent actions were added, add their preconditions as well.

Causal Link Protection: For every action A_t that might threaten a causal link $A_p \stackrel{R}{\rightarrow} A_c$ perform one of

(a) Demotion: Add $A_t < A_p$ to O'.
(b) Weak Promotion: Add $A_t \geq A_c$ to O'. If no agent can perform A_t concurrently with A_c, add $A_t > A_c$, instead.

If neither constraint is consistent, then return failure.

Non-concurrency enforcement For every action A_t that threatens a non-concurrency constraint $\mathsf{A} \neq A$ (i.e., A_t is an instance of the schema A that does not violate any constraint in B') add a consistent constraint, either

(a) Demotion: Add $A_t < A$ to O'.
(b) Promotion: Add $A_t > A$ to O'.

If neither constraint is consistent, then return failure.

Recursive Invocation: POMP($\langle A', O', L', NC', B'\rangle$,*agenda'*)

Figure 3: The POMP algorithm

Apart from handling conventional threats in a different manner, we have another form of threat in concurrent plans, namely, *NC-threats*. We say that action instance A_t *threatens* the non-concurrency constraint $\mathsf{A} \neq A_c$ if $O \cup \{A_t = A_c\}$ is consistent and A_t is an instantiation of A that does not violate any of the codesignation constraints. Demotion and promotion can be used to handle NC-threats much like they handle more conventional threats. Notice that although the set *NC* contains negative (inequality) constraints, they will ultimately be grounded in the set of positive constraints in O. Following the approach of [17], we do not consider an action to be a threat if some of its variables can be consistently instantiated in a manner that would remove the threat.

We can prove the following:

Theorem POMP is sound and complete.

That is, when POMP returns a plan for a particular goal and initial state, any n-linearization of that plan will reach the goal state from the initial state. Moreover, if there is a plan, POMP will generate it.[10] More specifically, if there exists a sequence of joint actions that achieves all goals, POMP will find a plan whose linearizations also ensure goal satisfaction.

When we compare POMP to POP (as applied to the joint action space) we see that POMP sometimes must make several more choices at a particular iteration of the algorithm—it must choose how to instantiate actions that must occur concurrently with a newly added action, and it must choose a threat resolution strategy for NC-threats, should the need arise. However, POP will have an exponentially larger branching factor in its action selection phase because it must choose among the set of joint actions.

We now sketch how POMP would solve the block moving problem mentioned in the introduction. In the initial state of our planning problem, a single block B is in on the floor in Room1, both sides of the table are down, and the two agents are in Room1. The goal is to have B on the floor of Room2 and both sides of the table down. The agents can pick up and put down the block, they can lift and lower each side of the table, and they can move the table. Precise action descriptions appear in a longer version of this paper, but their intuitive meaning should be clear.

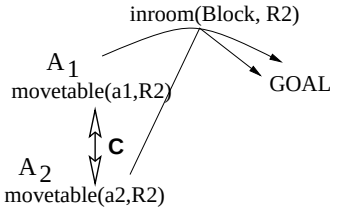
Suppose that *InRoom*$(B, Room2)$ is the first goal selected. This can be achieved by performing A_1 = *MoveTable*$(a1, Room2)$ via its conditional effect (note that $a1$ is an agent variable, so there is no commitment to which agent performs this action).[11] We must add both *Holding*$(a1, Table)$ and *OnTable*(B) to the agenda and insert

another concurrent action (see discussion in Section 2), we must use $O \cup \{A_p \leq A_t \leq A_c\}$ in the definition of threat, and we must change weak promotion to standard promotion.

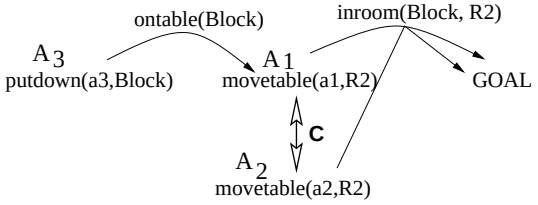
[10] Of course, in practice an appropriate search mechanism must "implement" the non-deterministic choice.

[11] We do not pursue the notion of heuristics for action selection here; but we do note that this action is a plausible candidate for selection in the multi-block setting. If the goal list asserts that a number of blocks should be in the second room, the single action of moving the table will achieve all of these under the appropriate conditions (i.e., all the blocks are on the table). If action selection favors (conditional) actions that achieve more goals or subgoals, this action will be considered before the actions needed for "one by one"

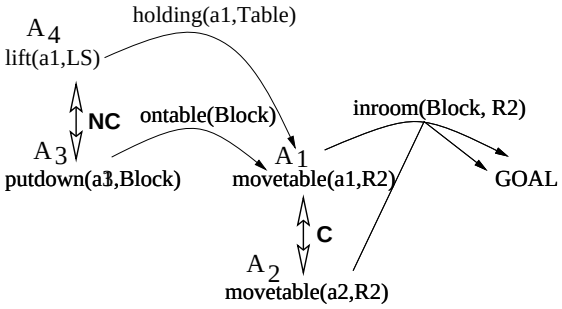
the appropriate causal links. In addition, the concurrent list (appearing in the conditional effect of the *MoveTable* action) forces us to add the action $A_2 = \textit{MoveTable}(a2, \textit{Room2})$ to the plan together with the non-codesignation constraint $a1 \neq a2$. The ordering constraint $A_1 = A_2$ is added as well. When we add A_2, we must add its precondition $\textit{Holding}(a2, \textit{Table})$ to the agenda as well. The structure of the partially constructed plan might be viewed as follows:



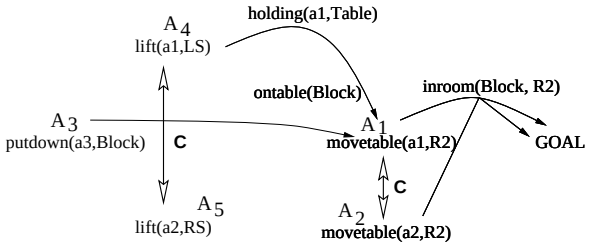
Next, we choose the just-added subgoal $\textit{OnTable}(B)$ from the agenda. We add action $A_3 = \textit{PutDown}(a3, B)$ to the plan with appropriate ordering constraint $A_3 < A_1$; its preconditions are added to the agenda and a causal link is added to L. In addition, we must add to *NC* the non-concurrency constraint $\textit{notLift}(a, s)$: no agent can lift any side of the table while the block is being placed on it if the desired effect is to be achieved. The new plan is shown below (we use left-to-right ordering of actions to denote temporal ordering of actions):



Now, we choose the subgoal $\textit{Holding}(a1, \textit{Table})$. This can be achieved using $A_4 = \textit{Lift}(a1, s1)$, with the ordering constraint $A4 < A1$. All the preconditions are added to the agenda, but no concurrency conditions are added (yet!), since we do not yet need to invoke the conditional effects of that action induced by simultaneous lifting of the other side of the table:



We now note that the conditional effect of A_4 poses a threat to the causal link $A_3 \overset{ontable}{\rightarrow} A_1$; this is because lifting a *single* side of the table will dump the block from the table. In addition, the non-concurrency constraint associated with A_3, that no lifting be performed concurrently with A_3, is threatened by A_4 (an NC-threat), as indicated in the plan diagram above. The *confrontation strategy* is used to handle the first threat, and action $A_5 = \textit{Lift}(a4, s2)$ scheduled concurrently with A_4.[12] The constraints $s1 \neq s2$ and $a4 \neq a1$ are also imposed. This ensures that the undesirable effect will not occur. We resolve the NC-threat by ordering A_3 before A_4.[13] The resulting partially completed plan is now free of threats:



After a number of additional steps, we obtain the final, successful plan represented in Figure 4. (We have ignored the initial picking up of the block by agent $a3$). One possible (compact) linearization of this plan is as follows: some agent $a3$ puts the block on the table (action A_3), which is then lifted concurrently by two agents ($a1$ and $a2$ perform A_4, A_5). Then these agents move the table concurrently (A_1, A_2) to Room2. One of them lowers its side of the table (A_6) resulting in the block falling to the floor. Then the other agent lowers its side as well (A_7).

We note that this plan does not commit particular agents to particular roles; e.g., the agent who puts the block on the table can be the one who picks up the right side or the left side of the table. Nor does the plan commit to the particular linearization described above (though any other linearization requires more "time"). Recall that a linearization consists of a sequence of *joint* actions. In the linearization described above, we assume an agent that is involved in no action from plan P at a particular point in time "executes" a no-op. Notice, however, that such no-ops simply serve as formal "placeholders" in the linearization; they do not appear in the plan, nor do they play a role in the planning process.

5 Concluding Remarks

Historically, planning with interacting actions was thought to be an inherently problematic affair. Thus, it is somewhat surprising that only minor changes are needed to enable the STRIPS action representation language to capture interacting actions, and that relatively small modifications to existing nonlinear planners are required to generate concurrent plans.

Our solution involves the addition of a concurrent action list to the standard action description, specifying which ac-

transport of the blocks by the individual agents. So this choice is not as silly as it might seem in the single-block setting.

[12] Confrontation is a threat removal strategy used in the context of conditional actions (see [17]).

[13] In anticipation of a subsequent step, we use variable $a2$ in the plan diagram instead of $a4$, since they will soon be unified. To keep things concrete, we have also replaced $s1$ and $s2$ with particular sides of the table, *LeftSide* and *RightSide*, to make the discussion a bit less convoluted.

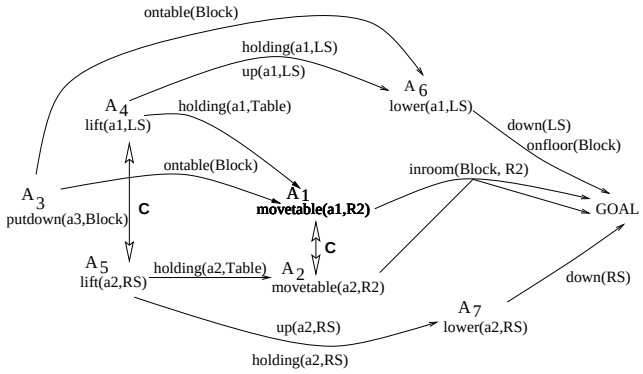


Figure 4: A Concurrent Nonlinear Plan

tions should or should not be scheduled concurrently with the current action in order to achieve a desired effect. There is a close connection between this type of specification and Knoblock's approach to generating *parallel execution plans* [11]. Knoblock adds a list to the action description that describes the resources used by the action: actions that require the same resource (e.g., access to a database) cannot be scheduled at the same time. Hence, Knoblock's resource list actually characterizes a form of non-concurrency constraints. In fact, certain non-concurrency constraints are more naturally described using such resource lists than with the general method proposed here. Augmenting our language with such lists is straightforward.

Apart from traditional nonlinear planners like UCPOP, newer planning algorithms, such as Graphplan [1] or Kautz and Selman's stochastic planning approach [9], can also be readily adapted to handle our multiagent representation language. In particular, Kautz and Selman's stochastic planner [9] can be viewed as using a one step planner as a subroutine. Such a planner is simple to construct whether one uses the standard STRIPS representation or our richer language. Once this planner exists, the algorithm behaves the same whatever the underlying language.

The approach we have considered is suitable for a team of agents with a common set of goals. It assumes that some central entity generates the plan, and that the agents have access to a global clock or some other synchronization mechanism (this is typically the case for a single agent with multiple effectors, and applies in certain cases to more truly distributed systems). An important research issue is how such plans can be generated and executed in a distributed fashion. This is an important question, addressed to some extent in the DAI literature, but for which adequate answers are still at large. Certain related issues are addressed in [2, 5]. The integration of classical planning with the more challenging aspects of multiagent systems (coordination, bargaining, etc.) should prove especially interesting [8].

Acknowledgments: This work was partially funded through IRIS project IC-7 and NSERC research grant OGP0121843.

References

[1] A. Blum and M. L. Furst. Fast planning through planning graph analysis. In *Proc. Fourteenth International Joint Conference on AI (IJCAI-95)*, pp.1636–1642, Montreal, 1995.

[2] C. Boutilier. Planning, learning and coordination in multiagent decision processes. In *Proc. Sixth Conference on Theoretical Aspects of Rationality and Knowledge (TARK-96)*, pp.195–210, Amsterdam, 1996.

[3] C. Boutilier and R. I. Brafman. Partial order multiagent planning and interacting actions. Tech. report, University of British Columbia, Vancouver, 1997 (forthcoming).

[4] C. Boutilier and M. Goldszmidt. The frame problem and Bayesian network action representations. In *Proc. Eleventh Biennial Canadian Conference on Artificial Intelligence*, pp.69–83, Toronto, 1996.

[5] R. I. Brafman and Y. Shoham. Knowledge considerations in robotics and distribution of robotic tasks. In *Proc. Fourteenth International Joint Conference on AI (IJCAI-95)*, pp.96–102, Montreal, 1995.

[6] T. Dean and K. Kanazawa. Persistence and probabilistic projection. *IEEE Trans. on Systems, Man and Cybernetics*, 19(3):574–585, 1989.

[7] B. R. Donald, J. Jennings, and D. Rus. Information invariants for cooperating autonomous mobile robots. In *Proc. Intl. Symp. on Robotics Research*, 1993.

[8] E. Ephrati and J. S. Rosenschein. Divide and conquer in multiagent planning. In *Proc. Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pp.375–380, Seattle, 1994.

[9] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. Thirteenth National Conference on AI (AAAI-96)*, pp.1194–1201, Portland, OR, 1996.

[10] O. Khatib, K. Yokoi, K. Chang, D. Ruspini, R. Holmberg, A. Casal, and A. Baader. Force strategies for cooperative tasks in multiple mobile manipulation systems. In *Int. Symp. of Robotics Research*, 1995.

[11] C. A. Knoblock. Generating parallel execution plans with a partial-order planner. In *Proc. Third Intl. Conf. on AI Planning Systems (AIPS '94)*, 1994.

[12] A. L. Lansky. Localized Event-Based Reasoning for Multiagent Domains. Tech. Report 423, SRI International, 1988.

[13] Y. Moses and M. Tennenholtz. Multi-entity models. *Machine Intelligence*, 14:63–88, 1995.

[14] J. S. Penberthy and D. S. Weld. Ucpop: A sound, complete, partial order planner for adl. In *Principles of Knowledge Representation and Reasoning: Proc. Third Intl. Conf. (KR-92)*, pp.103–114, Cambridge, MA, 1992.

[15] R. Reiter. Natural actions, concurrency and continuous time in the situation c alculus. In *Principles of Knowledge Representation and Reasoning: Proc. Fifth Intl. Conf. (KR-96)*, 1996.

[16] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation (Papers in Honor of John McCarthy)*, pages 359–380. Academic Press, San Diego, 1991.

[17] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, Winter 1994:27–61, 1994.